CS230

Splitting into train, dev and test sets

Best practices to split your dataset into train, dev and test sets

← Previous page Next page →

Splitting your data into training, dev and test sets can be disastrous if not done correctly. In this short tutorial, we will explain the best practices when splitting your dataset.

This post follows part 3 of the class on "Structuring your Machine Learning Project", and adds code examples to the theoretical content.

This tutorial is among a series of tutorials explaining how to structure a deep learning project. Please see the full list of posts on the main page.

Theory: how to choose the train, train-dev, dev and test sets

Please refer to the course content for a full overview.

Setting up the training, development (dev) and test sets has a huge impact on productivity. It is important to choose the dev and test sets from the **same distribution** and it must be taken randomly from all the data.

Guideline: Choose a dev set and test set to reflect data you expect to get in the future.

The size of the dev and test set should be big enough for the dev and test results to be representative of the performance of the model. If the dev set has 100 examples, the dev accuracy can vary a lot depending on the chosen dev set. For bigger datasets (>1M examples), the dev and test set can have around 10,000 examples each for instance (only 1% of the total data).

Guideline: The dev and test sets should be just big enough to represent accurately the performance of the model

If the training set and dev sets have different distributions, it is good practice to introduce a **train-dev set** that has the same distribution as the training set. This train-dev set will be used to measure how much the model is overfitting. Again, refer to the course content for a full overview.

Objectives in practice

These guidelines translate into best practices for code:

- the split between train / dev / test should **always be the same** across experiments
  - otherwise, different models are not evaluated in the same conditions
  - we should have a **reproducible script** to create the train / dev / test split
- we need to test if the **dev** and **test** sets should come from the same distribution

Have a reproducible script

The best and most secure way to split the data into these three sets is to have one directory for train, one for dev and one for test.

For instance if you have a dataset of images, you could have a structure like this with 80% in the training set, 10% in the dev set and 10% in the test set.

```
data/
    train/
        img_000.jpg
        ...
        img_799.jpg
    dev/
        img_800.jpg
        ...
        img_899.jpg
    test/
```

```
        img_900.jpg
        ...
        img_999.jpg
```

Build it in a reproducible way

Often a dataset will come either in one big set that you will split into train, dev and test. Academic datasets often come already with a train/test split (to be able to compare different models on a common test set). You will therefore have to build yourself the train/dev split before beginning your project.

A good practice that is true for every software, but especially in machine learning, is to make every step of your project reproducible. It should be possible to start the project again from scratch and create the same exact split between train, dev and test sets.

The cleanest way to do it is to have a `build_dataset.py` file that will be called once at the start of the project and will create the split into train, dev and test. Optionally, calling `build_dataset.py` can also download the dataset. We need to make sure that any randomness involved in `build_dataset.py` uses a **fixed seed** so that every call to `python build_dataset.py` will result in the same output.

Never do the split manually (by moving files into different folders one by one), because you wouldn't be able to reproduce it.

An example `build_dataset.py` file is the one used here in the vision example project.

Details of implementation

Let's illustrate the good practices with a simple example. We have filenames of images that we want to split into train, dev and test. Here is a way to split the data into three sets: 80% train, 10% dev and 10% test.

```
filenames = ['img_000.jpg', 'img_001.jpg', ...]

split_1 = int(0.8 * len(filenames))
split_2 = int(0.9 * len(filenames))
train_filenames = filenames[:split_1]
dev_filenames = filenames[split_1:split_2]
test_filenames = filenames[split_2:]
```

Ensure that train, dev and test have the same distribution if possible

Often we have a big dataset and want to split it into train, dev and test set. In most cases, each split will have the same distribution as the others.

What could go wrong? Suppose that the first 100 images (`img_000.jpg` to `img_099.jpg`) have label 0, the 100 following label 1, ... and the last 100 images have label 9. Then the above code will make the dev set only have label 8, and the test set only label 9.

We therefore need to ensure that the filenames are correctly shuffled before splitting the data.

```
filenames = ['img_000.jpg', 'img_001.jpg', ...]
random.shuffle(filenames)  # randomly shuffles the ordering of filenames

split_1 = int(0.8 * len(filenames))
split_2 = int(0.9 * len(filenames))
train_filenames = filenames[:split_1]
dev_filenames = filenames[split_1:split_2]
test_filenames = filenames[split_2:]
```

This should give approximately the same distribution for train, dev and test sets. If necessary, it is also possible to split each class into 80%/10%/10% so that the distribution is the same in each set.

Make it reproducible

We talked earlier about making the script reproducible. Here we need to make sure that the train/dev/test split stays the same across every run of `python build_dataset.py` .

The code above doesn't ensure reproducibility, since each time you run it you will have a different split.

To make sure to have the same split each time this code is run, we need to fix the random seed before shuffling the filenames:

Here is a good way to remove any randomness in the process:

```
filenames = ['img_000.jpg', 'img_001.jpg', ...]
filenames.sort()  # make sure that the filenames have a fixed order before shuffl
random.seed(230)
random.shuffle(filenames) # shuffles the ordering of filenames (deterministic giv
split_1 = int(0.8 * len(filenames))
split_2 = int(0.9 * len(filenames))
train_filenames = filenames[:split_1]
dev_filenames = filenames[split_1:split_2]
test_filenames = filenames[split_2:]
```

The call to filenames.sort() makes sure that if you build filenames in a different way, the output is still the same.

References

— course content

— CS230 code examples

← Previous page

Next page →

← BACK TO BLOG